SBIR Phase II Final Report
SBIR-12-03-3770 86-2

Release Date 5/8/92

NASA-17986
Sensor Frame, Inc.
Pittsburgh, Pa.

The Sensor Frame Graphic Manipulator NASA Phase II Final Report

NASA-CR-194243

PROJECT SUMMARY

PURPOSE OF THE RESEARCH:

Most of the useful information in the real world resides in humans, not in computers. Therefore we must find better ways of moving spatial information *from the human to the computer*. Quality 3-D graphics displays are necessary but *not sufficient* for a highly interactive and intuitive human interface. We need to improve input devices that capture human gestures and spatial knowledge.

One problem associated with direct manipulation interfaces in a design environment is that the user may not be skilled or precise enough to achieve the desired result. We can alleviate this problem through the use of *constrained virtual tools*. We define virtual tools as tools, displayed on the computer's video monitor, which are analogous to the tools used in factories, machine shops, or design studios. They include, but are not limited to, tools for cutting, smoothing, shaping, or joining operations. Virtual tools would map multifinger two and three-space gestures into the operations performed by the "business end" of the tool (such as the blade of a cutting tool), with constraints imposed by the model of the tool itself, the material or workpiece being operated upon, and the objectives of the user. The virtual tool would allow us to sculpt a smooth 3-D surface, varying the curvature or even the smoothness of a curve as it is drawn. However, the manipulation of a virtual tool requires *more* than six degrees of freedom. We believe that optical gesture recognition can provide up to twelve degrees of freedom per hand without the necessity for wires or gloves which inhibit casual use. The essential purpose of our research was to implement the enabling technology which makes casual use of virtual tools possible.

RESEARCH ACTIVITIES:

A prototype Sensor Cube was built using a neon-tube light source for contrast enhancement. A UNIX X-Windows interface was developed, and a control-panel builder was designed and implemented using X-Windows. A gesture-analysis package was developed, and is currently being extended for use in a multiple-finger environment.

RESEARCH RESULTS:

During the course of development of the three-dimensional Sensor Cube, we were informed that the sensors intended for use in the cube would no longer be available (see Section 3.1 for a more detailed discussion). This forced us to evaluate different approaches to optical multifinger sensing. Subsequently, we discovered a method of building the Sensor Cube with only one CCD sensor. This development will allow the three-dimensional Sensor Cube device to be less expensive than it's predecessor, the Sensor Frame. Unfortunately, the need to redesign the optical system and controller hardware and software of the cube delayed completion of this part of the project. Interesting and useful algorithms for 3-D finger tracking were developed and will be evaluated in detail as soon as sensor cube construction and interfacing are complete.

POTENTIAL COMMERCIAL APPLICATIONS:

The two-dimensional Sensor Frame technology will soon be supplanted by the three-dimensional capability of the Sensor Cube. However, the technology developed for use in the Sensor Frame has been transferred to a recently-announced commercial musical-instrument controller, the VideoHarp. The VideoHarp has attracted widespread attention in electronic-music circles, and was recently featured on the cover of Computer Music Journal (Volume 14, No. 1, MIT Press).

Sensor Cube gesture-recognition technology has it's greatest potential impact in computer-aided design (CAD) and teleoperation. Current input devices with six degrees of freedom or less are inappropriate for the manipulation of virtual tools. By gaining additional ability to capture the gestures of skilled scientists, designers, and technicians, computers will become a better alternative to traditional manual methods of design. If desktop manufacturing workstations with gesture-recognition input devices having up to 12 degrees of freedom can do for designers what time-sharing did for the programmers of the punch-card era, human productivity might be enhanced considerably; possibly by orders of magnitude.

1. Background and Motivation For Gesture-Based Systems

1.1. Virtual Reality and Virtual Tools

By the time a human child begins to speak, it has already spent approximately eighteen months to two years learning how to identify objects, people, and actions. It can distinguish one parent from another. It can distinguish itself from other objects and people. It can grasp and manipulate objects. Spatial knowledge comes early, and preceeds language.

Many young children can thread a nut onto a bolt before they go to school. A child less than four years old can do this. The task requires more than six degrees of freedom per hand (ie - positioning and orientation of the object in three-space plus a grasping operation), and implies that manipulation of twelve or more independent parameters is not unusually difficult for a young human.

In contrast, most workstations available today allow simultaneous manipulation of only *two* independent parameters, using a mouse. One *can* specify and manipulate representations of three-space objects with a mouse; but decomposing a six-parameter task into at least three sequential two-parameter tasks is not only counterintuitive, time-consuming, and error-prone; it is a waste of time if we can find a better way. By analogy, we could probably show that anything one can do using a keyboard can also be done using a telegraph key. But most of us would not exchange our computer keyboards for telegraph keys, despite the fact that the latter is cheaper, simpler, smaller, and standardized.

These considerations have prompted several researchers to attempt to improve workstation interfaces with a view toward accommodating human gesturing and tool-manipulation ability. In section 1.3, we will describe several systems which permit manipulation of objects in three dimensions. We will discuss their usefulness and their drawbacks, and ask how they might evolve in the future. While much of the published literature on 3-D input devices concentrates on the videogame-like ambiance of virtual reality, we will move the emphasis toward the idea of virtual tools, a subset of virtual reality that concerns itself with the development of more productive tools for use in design. Design and the need for redesign are among the most costly components in the production of high-technology products such as airplanes, rockets and space vehicles, and of low-tech mass-produced products such as automobiles.

1.2. Virtual Tools

One problem associated with direct manipulation interfaces in a design environment is that the user may not be skilled or precise enough to achieve the desired result. We can alleviate this problem through the use of *virtual tools.* We define virtual tools as tools, displayed on a workstation's video monitor, which are analogous to the tools used in factories, machine shops, or design studios. They include, but are not limited to, tools for cutting, smoothing, shaping, or joining operations. Virtual tools would map multifinger two and three-space gestures into the operations performed by the "business end" of the tool (such as the blade of a cutting tool), with constraints imposed by the model of the tool itself, the material or workpiece being operated upon, and the objectives of the user. The virtual tool would allow us to sculpt a smooth 3-D surface, varying the curvature or even the smoothness of a curve as it is drawn.

Virtual tools might be used to add material to a workpiece, to cut material, or to extrude it. The motion of a tool might be low-pass filtered, with filter-cutoff frequency of the filter being controlled,for example, by the distance between two fingers.

As we evolve hierarchies of virtual tools, designer productivity will hopefully increase. If we can significantly shorten design time, customization will be easier... and it is important to realize in this context that the higher-order goods of mass production, the machines that make other machines, are often highly customized tools, made in small quantities, but requiring many design iterations over their useful lifetime. As we build the virtual tools that cut design time, learning time for the designer will also be shorter, in relation to productivity. This is especially true if the designer can see "immediate feedback" on his or her latest design at low cost.

1.3. Related Research In Gesture-Sensing Technology

How can we best capture human gestures for intuitive manipulation of spatial objects? There are several different approaches to solving this problem. First, let's look at several currently-available devices:

- The DataGlove (VPL Systems)
- The Dexterous Hand Master (Exos)
- The Spaceball (Spatial Systems)
- The Flying Mouse (SimGraphics Engineering Corp.)

The DataGlove and Dexterous Hand Master (DHM) both sense finger-flexing motions. The DataGlove also senses hand position and orientation using a "Polhemus sensor" developed by McDonell-Douglas. The Polhemus sensor determines position and orientation of the hand using an externally-generated oscillating electromagnetic field. The version of the DataGlove with a Polhemus sensor has the advantage that it can sense relatively large-scale hand positions and orientations. Knowing position and orientation of the palm of the hand, one can use knowledge of finger-joint flexure to determine fingertip position, for use in grasping and tool-manipulation applications. In addition, by inserting piezoelectric transducers in the fingertips of the glove, one could conceivably provide some degree of touch feedback. Force feedback is a more difficult problem. The DHM has the advantage that its determination of finger-joint flexure appears to be considerably more accurate and repeatable than that of production DataGloves. It has the disadvantage that it does not currently provide hand position and orientation, although this could probably be implemented if market demand warrants it. Users of the DHM assert that it is lighter and less encumbering than it looks, although the time required to fit it to the hand seems to preclude casual use.

The use of glove-like sensors to sense gestures poses some problems. Currently, these devices use a cable to transmit data from the glove to the workstation, making casual use difficult. More later about the importance of casual use.

Hand (and consequently fingertip) position sensing (as opposed to detection of finger-joint flexure) requires the use of the relatively-expensive Polhemus sensor, and its use can be complicated by the presence and movement of ferrous metals in the vicinity of the sensor. A variation of the DataGlove developed by for Nintendo games, the PowerGlove, uses sonar devices mounted in the glove, but this severely constrains the orientation of the hand.

In the case of the DataGlove, unless each user has his own glove, a workstation supporting the device must have multiple gloves available in order to support left and right-handed persons with varying hand sizes. The same is probably true for the Exos device. Neither device yet provides sufficiently accurate and repeatable fingertip position information for use in a virtual tool environment. These latter considerations are an argument against the use of glove-like devices in a virtual tool (as opposed to virtual-reality) environment. Nevertheless, for many applications, we should expect them to provide a reasonably cost-effective solution.

The Spaceball is essentially a 3-D joystick. It is a ball slightly larger than a tennis ball, mounted in such a way as to make extended use very comfortable. The spaceball is excellent for positioning and orienting displayed 3-D objects, and for modifying one's view of a stationary object. It has good accuracy and repeatability. Because it functions like a joystick, it has some of the disadvantages that the joystick has relative to a mouse, and it has only six degrees of freedom. Six degrees of freedom are adequate for positioning and orienting objects, but more degrees of freedom are required to manipulate virtual tools. Once the tool is positioned, there are more things we must do to make it work, and that is the problem.

The Flying Mouse is a three-button mouse with a Polhemus sensor inside, designed so that it is easy to pick up. One can position and orient it in space, and then press the buttons. This is almost good enough for virtual tools, but not quite. For virtual tools, one might prefer the buttons to be more analog, ie - pressure sensitive. A nice thing about the Flying Mouse is that it can function as a normal 2-D mouse when on a tabletop, a convenient feature. The builder, Simgraphics Engineering Corporation, is well aware of the importance of the design and CAD markets, and emphasizes development of software necessary for the future "virtual tool" environment.

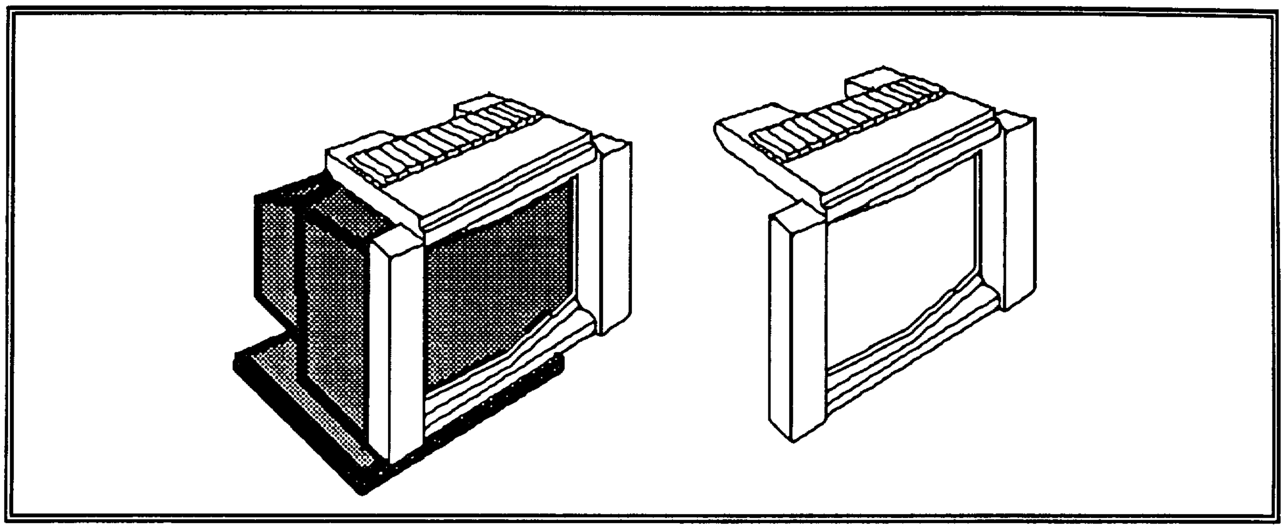
It is important to point out that the technologies we are describing are in their infancy, and constantly evolving. For this reason, many of the remarks pertaining to the products described above may become quickly outdated.

1.4. The Next Step: Vision-Based Gesture Sensing

The devices described in section 1.4 generally involve the use of mechanical, magnetic, or Hall-effect sensors in the sensing of palm position or finger flexure. A different approach to the problem of sensing multifinger gestures involves the use of vision-based systems.

Computer vision systems that analyze complex real-world scenes in real time remain beyond the state of the art. Nevertheless, in some applications, such as visual inspection, where scenes are specialized and predictable, systems are approaching feasibility (and a few systems are in commercial use).

At Sensor Frame Corporation in Pittsburgh, we have developed a device called a *Sensor Frame*, a 2-D optical finger-tracking device developed by the author and colleagues at Sensor Frame Corporation and Carnegie Mellon University. The prototype Sensor frame, using four sensors, reliably tracks up to three fingers at 30 Hz despite the fact that fingers sometimes block one-another from the point-of-view of some of the sensors. Tracking of *multiple* fingers is what distinguishes it from commonly-available touch screens. A drawing of the Sensor Frame, mounted on a monitor and in "standalone" mode, is shown below. The Videotape accompanying this report as Appendix C-1 shows the Sensor Frame in use.



The Mark IV Sensor Frame

Although the Sensor Frame represents a technology still in the early stages of it's development, it has aroused a fair amount of interest in industry, the press and media. In late 1988, CNN featured the Sensor Frame and VideoHarp in their AT&T Science and Technology series, and in 1989 Business Week featured both devices in their technology section. The Sensor Frame also appeared on the cover of NASA Tech Briefs, together with a feature article.

Unfortunately, production of the Sensor Frame, intended for September of 1989, was abruptly halted when the sensor manufacture halted delivery of optical dynamic-RAM sensors in the spring of 1989. This development is discussed in more detail in section 3.1. At present, we are developing the Sensor Cube, a 3D extension of the Sensor Frame, which will use one area CCD sensor to track up to three fingertips in three dimensions.

1.5. Future Applications of Gesture-Based Systems

Much of the motivation for building gesture-based systems can come from thinking about how we might apply them in the future in order to increase the productivity of designers. When we ask which gesture-sensing input devices will survive, we need to ask what future applications will require. Let's do a little thought experiment, and imagine what we would like our workstation to do for us if our objective were to design or modify a three-dimensional object, such as a machine-tool part, a piece of furniture, a molecule, or a nozzle for a rocket engine. We'll call this new type of workstation a *desktop manufacturing (DTM) workstation*, because it is intended to permit rapid prototyping of real-world objects. It would enable a designer to interactively specify or modify the shape of an object using spatial gestures and the *virtual tools* described above. Then it would build the object.

The DTM workstation would consist of the following components:

- A powerful CAD workstation that displays colored, shaded 3D objects, with full-motion video capability.
- A "3-D copier" similar to the stereolithography device manufactured by 3-D Systems Corporation. This device, or some future variation of it, will be used to fabricate a prototype or custom part quickly. There are currently at least three companies working on this aspect of DTM technology, and the number will probably increase.
- A 3-D gesture sensor, with gesture-recognition software and a *virtual-toolmaker's toolkit.*
- An optional 3-D laser scanner for scanning 3-D shapes.

2. Phase II Technical Objectives

Phase II Technical objectives consisted of the following:

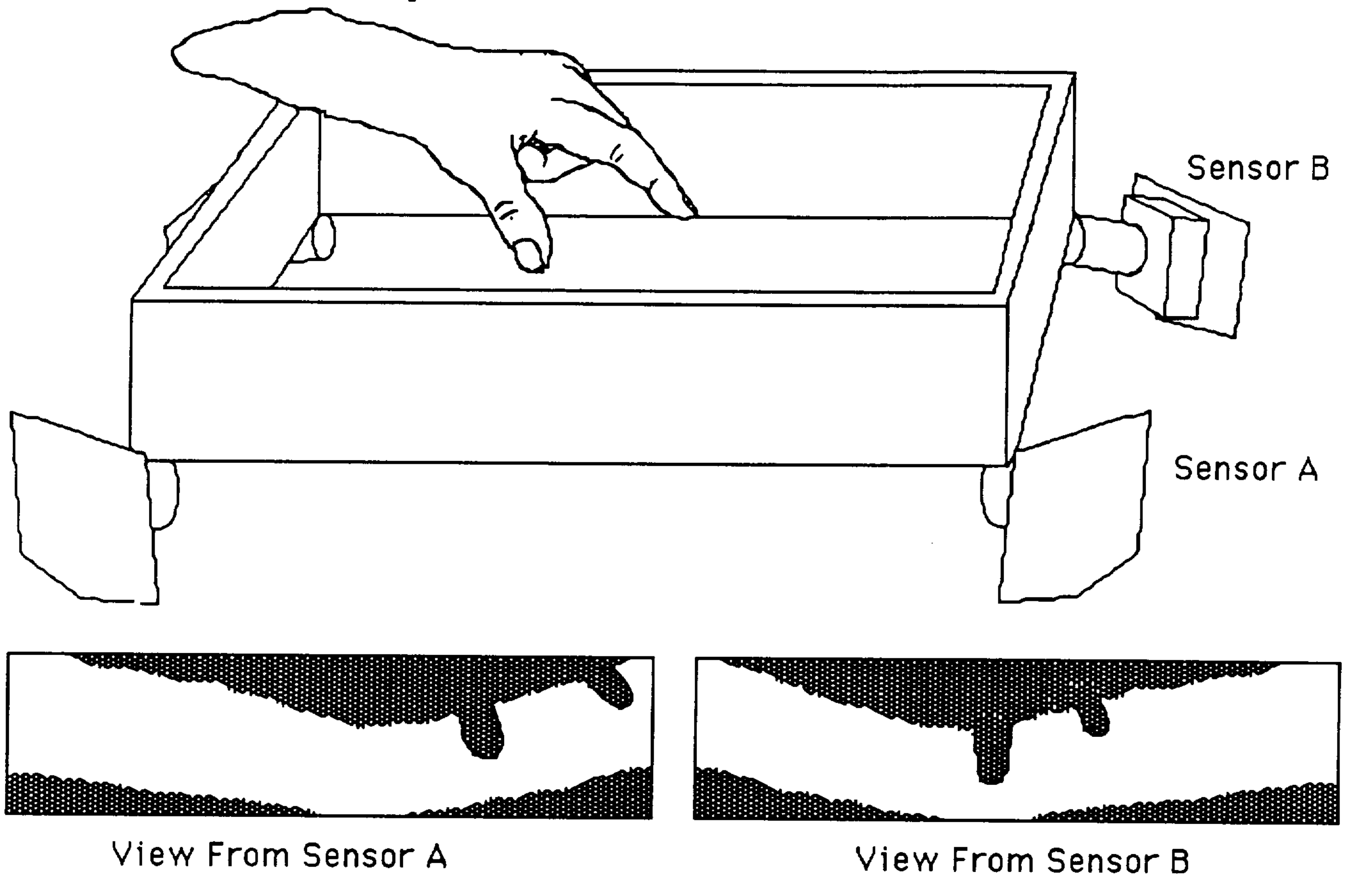
1. Development of Sensor Cube hardware and finger-tracking software.
2. Development of an intuitive interface for graphic-object manipulation.
3. Development of X-Window interface and UNIX device drivers for the Sensor Cube.
4. Development of soft control panels.

These objectives correspond to objectives 3.1.1 through 3.1.4, as described in our Phase II proposal for this project. Due to the sudden unavailability of DRAM sensors, as described in section 3.1 of this report, not all objectives were achieved in the form originally anticipated in the Statement of Work. Because the Sensor Cube design had to be modified significantly as a consequence of the sensor-availability problem, the resultant implementation delay precluded implementation of the 3D aspects of task 3.1.2.

3. Methodology, Observations, And Results

3.1. Development of Sensor Cube Hardware

In hardware terms, the *Sensor Cube* described in our NASA Phase II proposal was intended to be a thicker version of the Sensor Frame. A Sensor Cube was built with a 4.5" deep neon light source and four dynamic RAM (DRAM) sensors of the type used in the original Sensor Frame. This first Sensor Cube hardware was completed on schedule, about six months after the inception of Phase II. The first Sensor Cube prototype is shown schematically below, and in a videotape enclosed as Appendix C-2 of this report.



The First Prototype Sensor Cube, and Two Views From the Sensors

After completion of the first Sensor Cube prototype, work began on a UNIX interface for a Silicon-Graphics workstation, and at the same time for an X-Windows interface for an IBM RT workstation.

The UNIX and X-Window projects were essentially complete, in March of 1989, when Sensor Frame Corporation was abruptly informed by Micron Technology Corporation, the sole supplier of the DRAM sensors, that the fabrication of their line of DRAM sensors had been terminated. Our plans for commercial production of the Sensor Frame, intended to begin in August 1989, had to be abandoned. Both the then-current Sensor Frame and Sensor Cube designs made use of the 256K optical DRAMS supplied by Micron. All supplies of the 256K DRAMS had been committed to larger users by Micron before we were informed of the decision, leaving us with only five sensors; enough for our single prototype Sensor Frame, plus one spare. We were told that we would be able to obtain 100 of the smaller 64K DRAMs; however, we considered the 64K devices unsuitable for use either in a commercial Sensor Frame or in a Sensor Cube. Nevertheless, we bought the 100 64K devices, because we had a third product on the drawing boards that *could* use it; the VideoHarp.

It is perhaps relevant at this point to discuss the original reasons for the selection of DRAM sensors rather than charge-coupled devices (CCDs) as sensors, as well as the decision not to seek out another DRAM vendor to supply the optical DRAMs.

In 1982, when the first precursor of the Sensor Frame was built, CCDs were extremely expensive compared to DRAMs, with linear CCDs running in the thousand-dollar range. Further, CCDs require much more complex interface circuitry than do dynamic RAMs. In the early 80's, there were no integrated-circuit devices to provide the complex clock pulses, with their carefully-controlled slew rates, required by CCDs. Although integrated CCD clock and level-conversion chips became available in the mid-to-late 80's, the system cost of reasonable-quality CCDs is still considerably greater than the cost of optical DRAM chips. Further, the DRAM chips had several desirable properties that CCDs currently lack, one of the most important being addressability. In addition, it has not been any easier to obtain a second-sourced CCD than is was to obtain a second-sourced optical DRAM.

Although desperate, we were unable to convince Micron Technology to reverse their decision. They had little incentive to persue this still-relatively-small sensor market, having been awarded a virtual monopoly on the American DRAM market (along with Texas Instruments and IBM, the only remaining American DRAM manufacturers) by the U.S. Department of Commerce decision in 1988 to severely limit the importation of DRAMS from Japan.

As a consequence of this unfortunate event we did two things. First, since we could not manufacture Sensor Frames or Sensor Cubes, we decided to produce the VideoHarp, an optically-scanned musical instrument, which was the only one of the three products resulting from Sensor Frame technology that could make use of the available 64K DRAMs. Second, since we knew that we must switch to a different sensor technology after the 100th VideoHarp was built, and in order to build a commercial Sensor Cube (at present, we believe that it may be possible for a future VideoHarp and Sensor Cube to use the same area sensor), one of us (Paul McAvinney) attempted to find a way to build a Sensor Cube using fewer sensors. This effort succeeded shortly thereafter in the summer of 1989, when we developed a design using only one sensor and two mirrors.

The illustration below shows the a perspective view of the resultant single-sensor Sensor Cube mounted on a video monitor. Unlike the Sensor Frame, this design requires use of a gray-scale sensor such as a CCD or MOS area sensor. However, because it requires fewer sensors and associated optics, it will probably be cheaper to produce than the first Sensor Cube design. It's Z-axis depth will be about six inches, a significant improvement over the original design. In addition, the new design lends itself more easily to use as a two-handed teleoperation device. If two cubes are positioned side-by-side, they can share a controller.

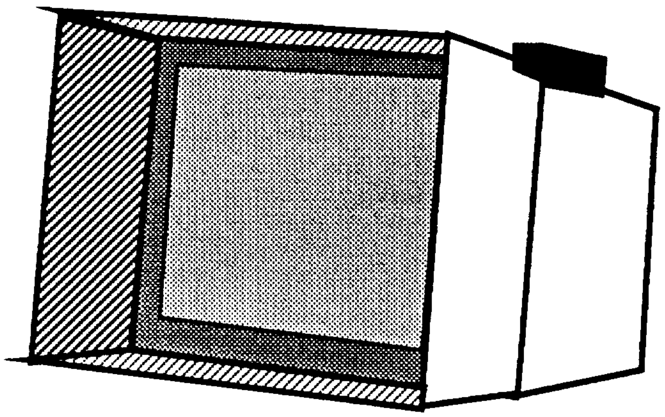
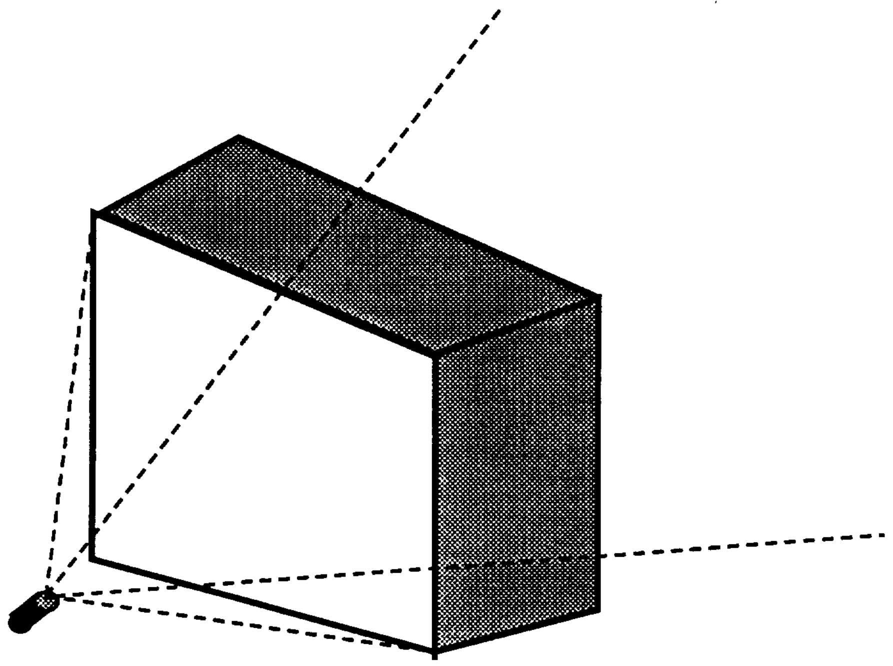


Figure 2: The Prototype Sensor Cube Mounted on a Video Monitor

Several important design considerations are driving the design of the second Sensor Cube. We list here the most important ones:

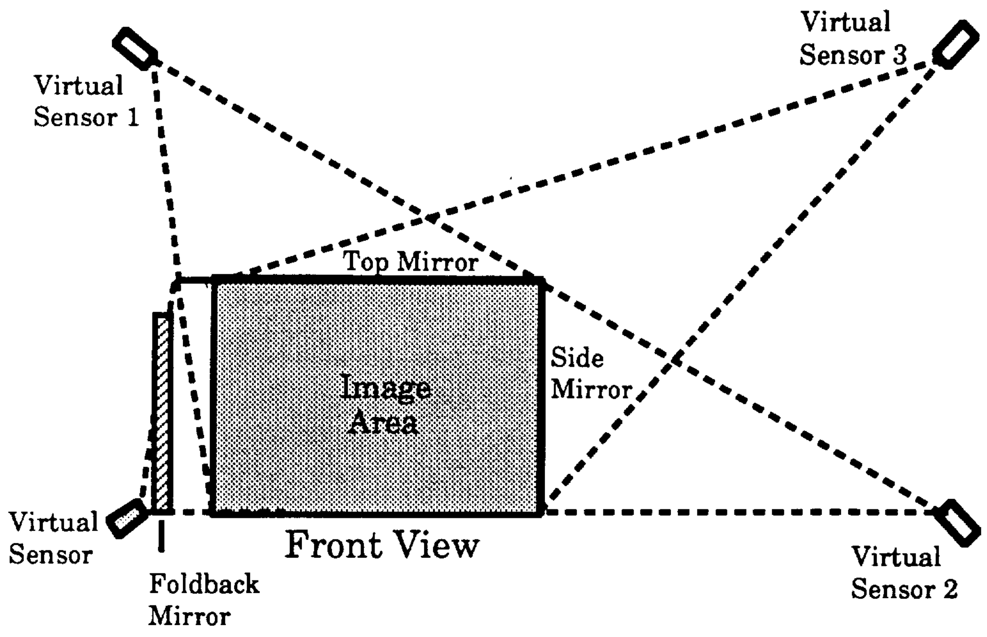
- The device must allow for at least ten degrees of freedom per hand, hopefully more. This will allow positioning and orientation of a virtual tool relative to a workpiece, followed by x-y manipulation of analog inputs on the tool itself by two opposed fingers. Even twelve degrees of freedom may not be too difficult to obtain.
- The device should allow casual use. This becomes especially important as increasingly powerful virtual tools permit a given operation to be completed in a short time, allowing the user to do something else which may not require the use of the gesture-sensing device. Good virtual tools should preclude the need for constant use, lessening concern about operator fatigue caused by holding one's hand in the air all day.
- The user's hands should be left free to use other devices, such as keyboards and telephones.
- Position of fingers relative to screen objects should be sensed.
- The device should be able to sense fingers in the vicinity of a video monitor. It should be attachable to the monitor, so that the user need not sacrifice desk space.
- It should operate independently of the video monitor, so that it can be mounted in another location (possibly for teleoperation-oriented applications) if the user so desires.
- It should be inexpensive in mass production, in order to encourage general use and standardization of application and user-interface software.

The next illustration shows a perspective schematic view of the new Sensor Cube design. The sensor is at lower left, and the shaded areas represent two mirrors at right angles.



Perspective View of Sensor Cube, With Monitor Screen At Rear

The next illustration shows the Sensor Cube from the front. Also shown are the positions of the *virtual sensors*. The scene produced in the single real sensor, at lower left, includes the scenes reflected from the mirrors along the top and right walls of the Sensor Cube enclosure. These virtual images may be treated geometrically as if they were images seen by the virtual sensors in the three positions shown. The net effect of the mirror system is to provide an image from four directions instead of just one. Since all sensors, real and virtual, look at the hand from a position near the plane of the video monitor, occlusion of fingers by the palm is minimized, except in the cases of extreme rotation of the hand.



Front View of Sensor Cube, Showing Virtual Sensors

The new Sensor Cube controller is currently under construction. Delays in delivery of support chips for the new design, based on a relatively inexpensive CCD designed by Texas Instruments, preclude the possibility of completion before the end of the NASA Phase II contract. Most U.S.- based CCD vendors have their CCD chips and support circuitry (and associated data sheets) produced in Japan for use in Japanese video cameras, and the U.S. wholesale market is small. As a consequence, some parts that have been on order for six months are still not being delivered.

A more long-term solution to the problem caused by the fact that there are currently no multiply-sourced area image sensors suitable for our designs is for us to design our own area sensor chip. This effort would make use of a scaleable CMOS process and the multi-foundry capabilities of the MOSIS prototyping service offered by the Information Sciences Institute at the University of Southern California (USC/ISI). PC-based software for MOSIS project-chip designs is available from commercial vendors at nominal cost.

Because of the uncertainty in the design schedule for this approach and our limited resources, we chose the more conservative approach of using commercial CCDs. Nevertheless, in the fall of 1989 we submitted a proposal to DARPA to fund a "smart" addressable MOS image sensor chip for use in gesture-based systems, but the proposal was rejected. We were told by DARPA that the proposal was considered technically sound, but that most if not all of their new funding had been reserved for HDTV and Star Wars projects. DARPA's approach may change, given the recent high-level shake-ups within the organization, but Sensor Frame Corporation intends to stake it's future on commercial product development (ie. - the VideoHarp), and fund new sensor development internally.

3.2. The Sensor Cube Finger-Tracking Algorithm

The algorithm for determining the spatial position and orientation of fingers in the Sensor Cube image area, stated here in somewhat oversimplified form, works as follows:

1) From the point-of-view of virtual sensor 3, the furthest sensor away from any sensed object, the scan line which intersects the mirrors at the greatest angle relative to the base-plane is read. This is guaranteed to sense a finger and allow it to be tracked at a z-axis value at or beyond the maximum guaranteed z-axis (Z_{MAX}) tracking value.

2) As any finger approaches Z_{MAX}, it is scanned by a "crosshair" pattern for each virtual sensor. One line of the crosshair is oriented *along the axis of the finger*, the angle being determined from previous scans. This is called the "longitudinal scan". For the simple case of a finger pointing directly along the Z axis, this value, taken from each virtual sensor, determines the position of the fingertip in the Z dimension. Information regarding fingertip position from each *longitudinal* scan is used to determine the height (above the fingertip) of the next *lateral* scan (see below).

3) The second scan is at right angles to the first, scanning across the *width* of the finger. This is called the "lateral scan". Information from each lateral scan is used to determine the lateral position of the next *longitudinal* scan.

In this method of tracking, each longitudinal scan corrects the position of the next lateral scan for a given finger, and vice-versa. Whether a frame-buffered image or an addressable sensor is used, the method allows us to locate fingers by scanning a relatively small fraction of the total number of pixels in the image, greatly reducing Sensor Cube controller processing requirements. In practice, two lateral scans of each finger may be needed to determine finger orientation accurately. When partial occlusion of a finger occurs, things become somewhat more complex. Experience with the Sensor Frame leads us to predict that we should not try to track more than three fingers at a time. This necessitates a style of gesturing which requires folding of the two smallest fingers into the palm. However, such a constraint appears to be easily learnable by most users. Further, the plane formed by three fingertips is useful for determining the orientation of a displayed object "grasped" by the hand. In the future, with more experience, we may try to relax the "three-finger" constraint.

3.3. Development of an Intuitive Interface for Graphic-Object Manipulation

Because the development of the Sensor Cube was delayed, the translation, rotation, grasping, and scaling of graphic objects in three dimensions was not possible. However, Appendix C-1, in the videotape attached to this report, shows how these capabilities were implemented using the Sensor Frame prototype for the two-dimensional case. We believe that when the Sensor Cube becomes operational, extension of these capabilities to the 3D case will not be difficult.

3.4. Development of an X-Window Interface and UNIX Device Drivers for the Sensor Cube.

X-Window and UNIX device-driver interfaces were successfully implemented for the Sensor Frame on IBM-RT and Silicon-Graphics IRIS workstations. The videotapes attached as appendices to this report show the effects of this implementation. Appendix B lists the implemented UNIX device-driver functions written in C.

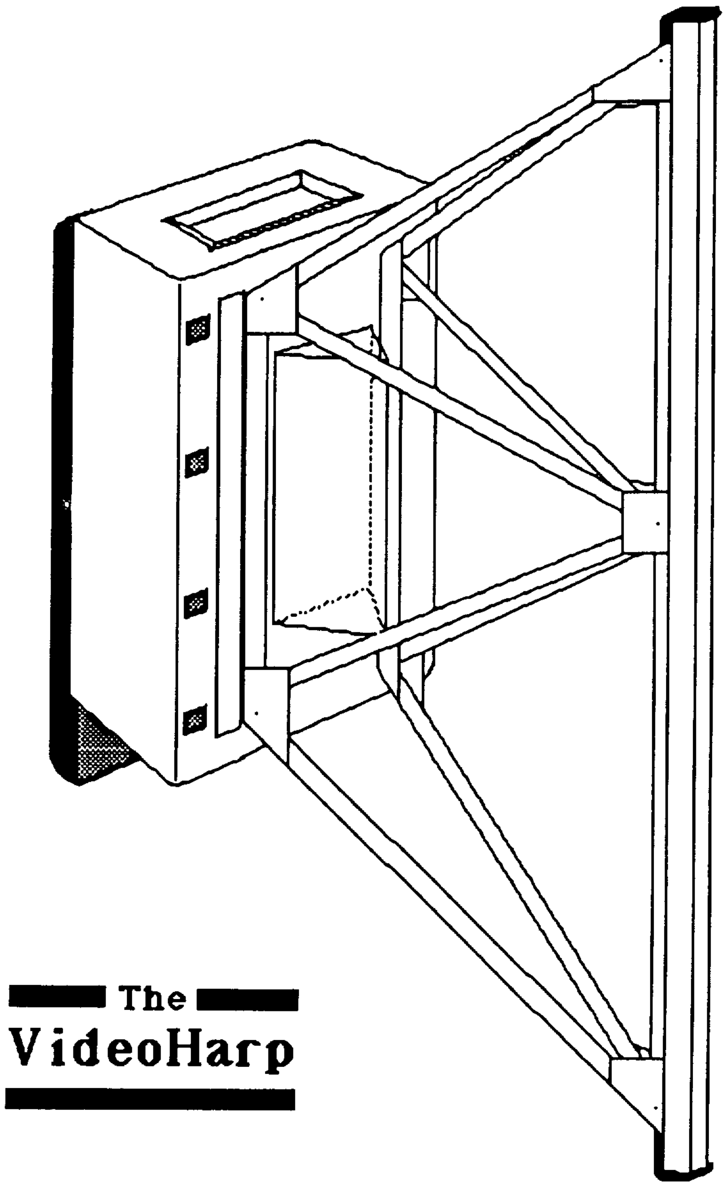
In general, it was found that the X-Windows interfaces (particularly on the IBM RT) were quite slow due to the excessive overhead of message passing between various X-Window components. This made multifinger tracking and screen update slow and difficult. The widely-acknowledged problem of excessive message-passing overhead has resulted in the recent appearance of terminals with processors dedicated to the efficient execution of X Windows.

Our implementation of the Sensor Frame on the Silicon-Graphics IRIS workstation was done using Sun's NEWS windowing system provided by Silicon Graphics.

3.5. Development of Soft Control Panels

The control-panel editing program was developed by researchers at Carnegie Mellon University under subcontract to Sensor Frame Corporation. An article describing this effort, "A Gesture Based User Interface Prototyping System", by Dr. Roger Dannenberg and Dale Amon of the School of Computer Science at Carnegie Mellon, was published in the Proceedings of the Second Annual ACM SIGGRAPH Symposium on User Interface Software and Technology, November 1989. That article is included in it's entirety as Appendix A of this report. The attached videotape (Appendix C-3) shows the operation of this system.

3.6. The VideoHarp

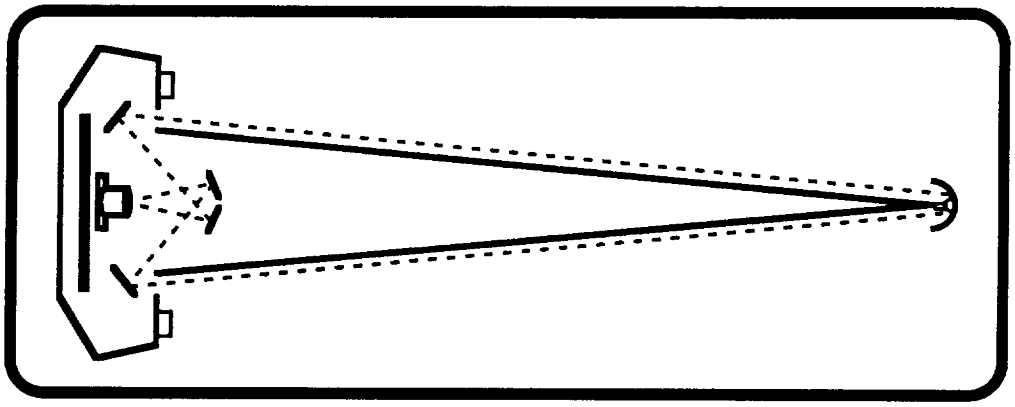


The VideoHarp is an optically scanned musical instrument which converts moving images of the fingers into music. Unlike keyboards and other mechanical sensing devices, the VideoHarp, because of the flexibility of it's optical scanning method, can recognize many classes of musical gestures, including bowing, strumming, keyboarding, and even conducting. A given class of gesture may be applied to any class of instrument timbre. For example, one could bow a horn or strum an organ. Using a fixed mechanical controller, such as a keyboard, one could produce non-keyboard sounds, such as the sound of a stringed instrument. However, even a keyboard with aftertouch cannot vary the timbre of a bowed note significantly as the note is played. Note that a cellist or violinist can press harder on the bow, play closer to the bridge of the instrument, and produce vibrato all at the same time. Keyboard controllers do not permit such quantitative, intuitive, and flexible control of many parameters at once.

The VideoHarp, because it can optically track all the player's fingers at once, allows control of many independent parameters. It permits a richness of timbral expression approaching, and often exceeding, that of traditional instruments. The playing surfaces of the VideoHarp can be divided up into *regions*. Each region possesses it's own attributes, such as which instrument is to be played, the width of keys, pitch and amplitude ranges, and many others too numerous to mention here.[1]

Some classes of gestures lend themselves well to *conducting*. For example, a stored score can be conducted using bowing motions. Each reversal of the bow causes the next note to be played. While one hand executes the bowing motions, the fingers of the other hand can be used to control additional aspects of timbre at the orchestral level[2]. This allows a novice user to obtain immediate musical results, and to express himself musically at a high level without having to learn all the nuances of the instrument. It is the *musical expression* which is important here, not the ability to specify which notes are to be played. That has already been done by the composer. Although a conductor may look at an orchestral score in order to plan what to do next, he is primarily interested in developing his own individualized expression or interpretation of the composition.

The following diagram illustrates the internal structure of the VideoHarp, as seen from above. The dashed line shows the light path from the light source (at right) to the sensor, at left. Mirrors are used to bend the light path so that both playing surfaces can be scanned by a single area sensor. Fingers placed against the playing surface block light from the light source, creating a shadow image on the sensor after being focused by a lens system (the cylindrical object at left).



The V2 VideoHarp, As Seen From Above

The VideoHarp can assume four different roles:

- **A Musical-Instrument Controller:** The VideoHarp is an optically-scanned free-hand gesture sensor adapted to the needs of the instrumentalist. It can be connected to any synthesizer with a MIDI input.

---

[1] For more information, see *The VideoHarp*, in Proceedings of the 14th International Computer Music Conference, Cologne Germany, 1988, Ed. Lishka and Fritsch.

[2] An orchestra can be thought of as a large instrument played by a conductor. The conductor does not specify the notes to be played, only *how* they are to be played.

- **A Conducting Controller:** The VideoHarp can capture gestures used in *conducting* a group of instruments, such as a quartet, an ensemble, or even a full orchestra.

- **A Composition Tool:** With it's ability to optically sense playing and conducting gestures of many types, the VideoHarp is an *enabling technology* which permits composers to experiment with the interaction between melody, tempo, timbre, and dynamics, with a flexibility and immediacy unmatched by current controllers.

- **A Complete Musical Instrument:** In the future, a VideoHarp with built-in synthesizer will be a complete musical instrument. At present, because there is no "standard" synthesizer, it is better to leave the choice of this device up to the user.

The VideoHarp has received national and international coverage in several publications, including Science News and Business Week. A color picture of the VideoHarp appeared recently on the cover of Computer Music Journal, which included a paper by the inventors.

4. Conclusions And Recommendations

We beleive that in the long run, vision-based gesture recognition systems such as the Sensor Cube will be widely used; first in design workstations, and later in personal computers, when full-motion video display of virtual tools and workpieces becomes inexpensive (this may happen relatively soon). We believe this for the following reasons:

- Casual, hands-free use of virtual tools will become increasingly important to users as the number, quality, cost, and utility of constrained virtual tools continues to shorten the design process and increase the number of people who will make use of it.
- Desktop Manufacturing (DTM) will allow fast prototyping, quick redesign, and inexpensive small-batch production of evolving products. As DTM becomes cheaper, a wider base of users will insist on standardized and portable virtual tools.
- Because each virtual tool must contain a description of the gesture-to-toolblade mapping, optical, rather than mechanical methods of gesture sensing permit the most flexible and repeatable interpretation of gestures having on the order of twelve degrees of freedom from a wide range of human hand and finger shapes.
- The Sensor Cube will be inexpensive in large quantities, and unobtrusive in casual use.

In the short run, we have to survive; we have had our problems obtaining a reliable supply of appropriate sensors and support circuits in a sensor market still dominated by video cameras for television applications. This situation has delayed construction of the Sensor Cube prototype (see Section 3.1), but things will probably improve. One Japanese image-sensor manufacturer has already requested that we submit a detailed proposal to them outlining our design requirements for a "smart" addressable area sensor. American IC manufacturers continue to lag in their understanding of the future role and importance of smart optical sensors which can detect and flag the pixel locations of image changes in the time domain.

We believe that the next great revolution in human productivity will be the result of a nonlinear increase in the utility and productivity of design tools. Good tools will make design more fun, and human creativity and productivity always profit when a process is viewed as being fun rather than work.

Appendix A: Reprint of Dannenberg/Amon SIGGRAPH Article

A Gesture Based User Interface Prototyping System

Roger B. Dannenberg and Dale Amon

School of Computer Science
Carnegie Mellon University
email: Roger.Dannenberg@cs.cmu.edu

Abstract

GID, for Gestural Interface Designer, is an experimental system for prototyping gesture-based user interfaces. GID structures an interface as a collection of "controls": objects that maintain an image on the display and respond to input from pointing and gesture-sensing devices. GID includes an editor for arranging controls on the screen and saving screen layouts to a file. Once an interface is created, GID provides mechanisms for routing input to the appropriate destination objects even when input arrives in parallel from several devices. GID also provides low level feature extraction and gesture representation primitives to assist in parsing gestures.

1. Introduction

Gestures, which can be defined as stylized motions that convey meaning, are used every day in a variety of tasks ranging from expressing our emotions to adjusting volume controls. Gestures are a promising approach to human-computer interaction because they often allow several parameters to be controlled simultaneously in an intuitive fashion. Gestures also combine the specification of operators, operands, and qualifiers into a single motion. For example, a single gesture might indicate "grab this assembly and move it to here, rotating it this much." Previous work on gesture based systems [1, 2, 6, 4, 12] has only begun to explore the potential of gestural input. We need a better understanding of how to construct gestural interfaces, and we need systems that allow us to prototype them rapidly in order to learn how to take advantage of gestures. Our work is a step toward these goals.

Building interactive systems based on gesture recognition is not a simple task. As we designed and implemented our system, we encountered several problems which do not arise in more conventional mouse-based systems. One problem is supporting multiple input devices, each of which might have many degrees of freedom. Unlike most mouse-based systems which can only engage in one interaction at a time, our system supports, for example, turning a knob and flipping a switch simultaneously.

Another problem is how to parse input into recognized gestures. We assume that gestures are specific to various interactive objects. For example, a switch displays an image of a toggle on the screen and can be "flipped" by a fingertip, but only if the finger travels across the image in the right direction. In this case, finger motion must be interpreted in the context of the interactive object, and a path (as opposed to instantaneous positions) defines the gesture.

Beyond these problems, we were also interested in making our prototyping environment easy to use, modular and extensible. Thus, we have been concerned with the issues of how to combine interactive objects in a screen-based interface, how to edit the layout and appearance of the interface, and how to encapsulate the behaviors of interactive objects and isolate them from other aspects of the system.

A final issue is the question of debugging support to aid in the implementation of new interactive objects. We use input logging to make bugs more reproducible and a combination of interpreted and compiled code to speed development.

We have completed a system, named GID for Gestural Interface Designer, in which one can interactively create and position instances of interactive objects such as menus, knobs and switches. One can interactively attach semantic actions to these objects. GID supports input from both a mouse and a free-hand sensor that can track multiple fingers. We are far from having the ultimate gesture based interface support environment, but we have developed interesting new techniques that are applicable to future gesture-based systems.

In section 2 we describe the structure of our prototyping system, and section 3 describes the handling of input from multiple devices. In section 4 we describe our general technique for processing input in order to recognize gestures. Section 5 describes in greater detail our development techniques and the current implementation. Conclusions are presented in section 6 along with suggestions for future work.

2. The Interface Designer

This project extends an earlier effort called Interface Designer, or ID. The goal of ID was to provide a small, practical and portable system for creating screen-based interfaces by direct manipulation. ID was inspired by Jean-Marie Hullot's work at INRIA, a precursor to Interface Builder [5, 9]. A typical use of ID might be the following: by selecting a menu item, the user creates an instance of an object which displays a 3-D database. In order to manipulate the image, the user creates a few instances of sliders. A short Lisp expression is typed to supply an action for each slider, and labels of "azimuth", "altitude" and "pitch" are entered. Now, moving a slider causes a message to be sent to the display object and the image is updated accordingly.

The basic internal structure of ID introduces no significant improvements over other object-oriented event-driven interface systems such as MacApp [11] or Cardelli's user interface system [3]. It will be described here, however, for clarity.

ID represents the screen as a tree of objects. At the root is a screen object that contains a set of window objects. Each window object may contain a set of control objects. One type of control object is the control group, which serves to collect a set of control objects into an aggregate. Other types of control objects include sliders, buttons and switches of various styles. (See figure 2-1.)

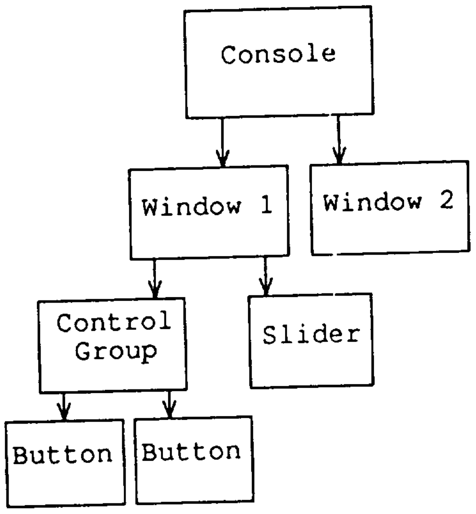


Figure 2-1: An ID control object tree.

In addition to the hierarchy implied by this tree, there is also a class hierarchy arranged so that classes can inherit much of their behavior. (See figure 2-2.) The InputControl class encapsulates generic behavior of objects that handle input from the user and manage some sort of image on the screen. PictureControls, a subclass of InputControls, actually draw images. These include classes such as Switch and Slider. Another subclass of InputControl is ControlGroup, which implements the search for an input handler. New interactive controls are typically created by subclassing PictureControl or one of its subclasses. Output-only "controls" have also been defined as subclasses of Control. For example, class 3dPict draws a wire-frame rendering of a 3-D data base which is loaded from a file.

```
Object
    Control
        InputControl
            PictureControl
                Button
                Switch
                Slider
                FontDev
            ControlGroup
                Console
                Window
                Menu
                MenuCard
                PictureGroup
            MenuItem
        OutputOnlyControl
```

Figure 2-2: Interface Designer class hierarchy.

In normal operation, ID has a single main loop that waits for input and delivers it to the appropriate destination. Each input event is represented by a window identifier, a device type (e.g. mouse or keyboard), coordinates (if any), and other data. This event is passed to the root of the tree where a search for a recipient begins. Typically, each node which is not a leaf node (a PictureControl) passes the event to each of its children until one of them accepts the input event.

To make this recursive search reasonably efficient, a ControlGroup object rejects mouse input which falls outside of its bounding box, and windows reject input unless the event's window identifier matches. Even with these optimizations, it is too inefficient to search the object tree from the root for each mouse-moved event during a dragging operation. Instead, a context mechanism is used.

In ID, the handler for input is found at the top of a *context stack*. An object can grab future input events by pushing a new context onto the stack to direct future input to the object. For example, a dragging operation would start with a mouse-down event that would be handled in the normal way. Upon receiving the mouse-down event, the object that handles the dragging operation pushes the context stack and becomes the target of future input. All successive mouse-move events go directly to the object. When mouse-up is received, the object pops the current context to restore input processing to normal.

The context stack has two uses in addition to temporarily grabbing mouse input. The context stack is used for nested pop-up windows and also for implementing an "edit" mode in which control objects can be created, moved, copied, and deleted. In edit mode, we want to be able to select controls without invoking their normal operations. This is accomplished by pushing a special "edit context" which routes all input to an editor that can manipulate the on-screen objects.

In some cases, one might want to effect a global context change, such as a pop-up dialog box which preempts all controls. This is accomplished by pushing a new context on the stack. This may redirect input from an object with a gesture in progress. We avoid problems here by sending a "finger up" event to the old handling object and a "finger down" event to the new handling object whenever a finger changes windows.

3. Parallel Input Handling

We used ID as the basis for GID, our gesture-based system. GID was designed to be used with a Sensor Frame [7] as the gesture sensing device. The Sensor Frame tracks multiple objects (normally fingers) in a plane positioned just above the face of a CRT display. The "plane" actually has some thickness, so three coordinates are used to locate each visible finger. When a finger enters the field of view, it is assigned a unique identifier called the *finger identifier* Each time the finger moves, the new coordinates of the finger and the finger identifier are transmitted from the Sensor Frame to the host computer. Ideally, when a finger enters the field of view of the Sensor Frame, it is assigned a number which it retains for the entire time it remains in view. Since the Sensor Frame may be tracking multiple fingers in parallel, coordinate changes for several fingers may be interleaved in time.

In our gesture-based system, we wanted be able to handle multiple finger gestures acting on a single object, for example, turning a knob. We also wanted to allow users to operate a control with each hand. The stack-based context mechanism described in the previous section, however, does not allow inputs to be directed to several objects. We could simply pass all input to the root of the object tree, but again, the search overhead would be too high.

Our solution is to maintain a more general mapping from input events to objects. Each context contains a list of input templates, each of which has an associated handling object. Input templates consist of a window identifier, device type, and finger identifier. If all elements of the template match corresponding elements of an input event (the template may have "don't care" values) then the event is sent to the indicated handling object. If no template matches, then input is sent to a default handling object, also specified in the current context. As a result, we can have:

- two fingers operating a knob (input from either finger is forwarded immediately to the knob object),
- another finger moving toward a switch (input from this finger goes to the root of the object tree as usual. The switch object may change the current context and take future input directly when the finger gets close), and
- a simultaneous mouse click on a button (this input would work its way through the object tree from the root to the button object).

4. Gesture Representation and Processing

Since individual finger coordinates do not convey any dynamic aspects of gestures, the first stage of processing Sensor Frame input data is to represent the path of each finger by a set of features. The features are then interpreted by controls. The current set of features includes a piece-wise linear approximation of the path, the point where the path first crosses into an "activation radius", and the cumulative angular change.

4.1. Initial Processing

The x,y,z coordinates are supplied by the Sensor Frame as integers but are translated to floating point for further processing. The x,y,z portion of the input data is referred to hereafter as a *Raw Data Point* or *RDP*.

Normally, the default handling object for RDP's is the root of the object tree. The tree is searched after each input; however, when the RDP falls within the bounding box of a control object, the object responds by putting a template in the current context that will direct future events with the same finger identifier to the object. Future matching events will arrive at the object where they are added to a table associated with both the object and the finger identifier. This table of RDP's is called an *open vector*.

4.2. Path Decomposition

The next step is to process the open vector of RDP's to obtain a segmented[1] representation. This representation simultaneously provides data reduction and immunity from jitter.

For convenience, we want our approximation to be continuous; that is, each segment begins where the previous one ended, and all endpoints coincide with data points (RDP's). The algorithm for constructing the approximation is straightforward: as each RDP is added to the open vector, and error measure is computed. When the error measure exceeds a constant threshold, a segment from the first to the next-to-last point is added to the path and the open vector is adjusted to contain the last two RDP's. This algorithm can be described as "greedy without backtracking" since we pack as many RDP's into each segment as possible (limited by the error threshold) and we

[1] In this discussion, a *segment* is an ordered pair of points, e.g. RDP's, and a point is an x, y, z triple.

never try alternative assignments of RDP's to segments.

Figure 4-1 illustrates the process. The segment from point 1 to point 3 falls below the error threshold, but a segment from point 1 to point 4 exceeds the threshold. Therefore, the segment [point 1, point 3] is added to the path, and a new open vector [point 3, point 4] is started. This is extended to point 5 and then to point 6.

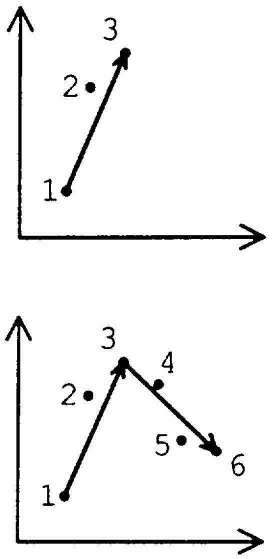


Figure 4-1: Fitting vectors to a set of points.

The error measure is:

$$error = \frac{1}{n}\sqrt{\left(\sum_{i=1}^{n}|D_x(p_i)|\right)^2 + \left(\sum_{i=1}^{n}|D_y(p_i)|\right)^2 + \left(\sum_{i=1}^{n}|D_z(p_i)|\right)^2}$$

where $D_x(p_i)$ is the x-component of the shortest vector from an RDP p_i to the proposed segment $[p_1, p_n]$ from point p_1 to p_n. We elected not to take a sum-of-squares in the innermost summation to save a bit of computation, and the resulting path decomposition seems to work well. The distance from a point to a line can be computed without trigonometric or square root functions as shown in Appendix I.

4.3. The Activation Volume

Gesture analysis is performed if an open vector passes into the volume defined by an activation radius and an activation center. Such processing will continue so long as succeeding RDP's remain within that volume.

An activation center is not necessarily static. For example, the knob on a slider has an activation center that moves along with it. The value associated with the device class is in this case a default initial value for the slider location.

Because we are polling the Sensor Frame from the application program, we cannot guarantee that we will catch all (or any) relevant RDP's within a possibly small *activation volume*. This is particularly true if the finger is traveling quickly. However, by setting the size of the bounding box large enough, we can guarantee we will at least pick up endpoints of a path segment that intersects this volume. The same distance algorithm (see Appendix I) used for path decomposition is then used to see if the point of closest approach of the path to the *activation center* is less than the *activation radius*.

4.4. Gestures

Once an RDP falls within the activation radius, the gesture features are examined by the corresponding object. Response to gestures is programmed procedurally for each type of control.

A toggle switch (or any other control affected by a simple linear motion), can be moved if the direction of travel of a finger path (A) matches the preferred axis of travel of the device (B). We define a maximum angle (θ_{max}) between the two and see if the actual angle (θ_{act}) is within bounds.

The actual angular error can be found using the definition of the vector dot product:

$$\mathbf{A}\cdot\mathbf{B} = |A||B|\cos(\theta_{act})$$

and rearranging to solve for $\cos(\theta_{act})$:

$$\cos(\theta_{act}) = (\mathbf{A}\cdot\mathbf{B})/(|A||B|)$$

If the inequality:

$$\cos(\theta_{act}) \le \cos(\theta_{max})$$

holds, then the movement of the finger is close enough to the preferred direction to cause a state change. Note that $\cos(\theta_{max})$ is a constant that can be precalculated, thus we avoid calculating transcendentals at run time by comparing cosines of angles instead of the angles themselves and by using the equation:

$$\mathbf{A}\cdot\mathbf{B} = A_xB_x + A_yB_y + A_zB_z$$

The knob rotation gesture consists of one or two fingers moving within the activation radius of the knob. Once it is determined that a finger path crosses the activation radius, an angle from the center of the knob to the finger is computed and saved. Each location change within the activation radius results in a recalculation of the angle, and the angle of the knob is updated by the angular difference. When there are two fingers within the activation radius, the knob is updated when either finger moves; the overall knob rotation is effectively the average rotation of the two fingers.

5. System Considerations

5.1. Implementation Languages

Our Sensor Frame interface, gesture recognition software, and graphics primitives are all implemented in the C programming language. Graphical and interactive objects, as well as the top-level input handling routines, are implemented in XLISP, a lisp interpreter with built-in support for objects.

Although we would have preferred a compiled lisp, this work was begun at a time when our workstation environment was in a state of rapid change. During the course of the project, we ported XLISP to three machine types and implemented our graphics interface on two window managers. The fact that XLISP is a relatively small C program made it easy to port and to extend with the additional graphics and I/O primitives we needed.

5.2. Input Diagnostics

For diagnostic purposes, input of raw position data points is done through a device-independent module that allows input to come from a Sensor Frame, to be partially simulated by a mouse, or to be played back from a file that was "recorded" on a previous run with a mouse or a Sensor Frame. Bugs that appear only in long runs can be reproduced by playing back the log file during a debugging session.

The interface is implemented in such a way that regardless of which device is being used as the pointing device, the window menu is still available via the mouse. Commands are available to display every RDP as a small box on the screen; to print the results of every Sensor Frame input to a diagnostic window; to select a prerecorded file, a mouse or the Sensor Frame as the source of input; or to begin or end recording data for future playback.

6. Results and Conclusions

In the process of building GID, we have encountered several problems which are worth further study. One problem is how to organize prototyping software such as GID to allow controls to be operated in "run" mode and edited in "edit" mode. It seems inappropriate to implement editing within each object (Should a slider contain code for editing its size, placement, label, etc?), but a modular approach is preferable to a monolithic editor that captures all input in edit mode. In GID, we divert input when in "edit" mode, but we have specific editing methods in various subclasses of Control. One alternative is to implement all interactive behavior outside of control objects as in Garnet [8].

Another problem is that we have no high-level procedures for recognizing complex gestures: our recognizers must be hand-coded using fairly low-level representations. A promising alternative is the pattern recognition approach being pursued by Dean Rubine [10].

We know of no window managers that support multiple cursors. Ideally, the window manager should track each finger with a cursor and also determine what window contains each visible finger. Currently, the overhead of cursor tracking and mapping input to windows from outside of the window manager (X11) causes significant performance problems.

The present resolution of the Sensor Frame is only about 160 x 200 points. While this provides plenty of resolution relative to the size of controls displayed on the screen, greater resolution is needed in order to accurately measure the direction of motion and to minimize jitter.

The organization of GID prevents a single gesture from being received by multiple controls simultaneously. We do not feel this is a serious limitation, but it could be avoided by utilizing a more complete mapping from RDP's to objects. Rather than searching the object tree depth first, we could use hashing or a linear search of all objects to locate potentially overlapping bounding boxes which contain each RDP. Input events would then be duplicated and sent to each "interested" object. This technique was tried in an earlier system and allowed, for example, two adjacent switches to be flipped by moving a finger between them.

We note that some window managers might assist in the implementation of controls: if each control is implemented as a sub-window, then the search for a handler could be performed by the window manager. This technique will *not* work if we want input to reach multiple controls because current window managers will map input to only one window even if there is overlap. Furthermore, window managers typically assume a single pointing device, and extensive modification would be required to handle input from the Sensor Frame or some other gesture sensing device.

In conclusion, we have implemented a system for prototyping gesture-based user interfaces. The system is capable of editing its own interface, and applications are typically built by extension. The system allows us to experiment with screen layout and with multiple input devices without programming, and the system is extensible so that new interaction techniques can be integrated and evaluated. We have found piecewise linear approximations to paths to be an appropriate representation for simple gestures, and our vector software can be reused by different control objects.

7. Acknowledgments

We would like to thank Paul McAvinney, who played a crucial role in this research as inventor of the Sensor Frame and promoter of the value of gestures. Portions of this work were supported by the Sensor Frame Corporation as part of a project sponsored by NASA. Workstations used in this study were made available through an equipment grant from IBM. Thanks are also due to to Gerald Agin for the time he spent discussing curve fitting algorithms and analytic geometry.

References

1. R. A. Bolt. *The Human Interface: where people and computers meet.* Lifetime Learning Publications, 1984.

2. Frederick P. Brooks, Jr. Grasping Reality Through Illusion - Interactive Graphics Serving Science. CHI '88 Proceedings, May, 1988, pp. 1-11.

3. Luca Cardelli. Building User Interfaces by Direct Manipulation. Tech. Rept. 22, Digital Equipment Corporation Systems Research Center Research Report, Oct., 1987.

4. S. S. Fisher, M. McGreevy, J. Humphries, and W. Robinett. Virtual Environment Display System. ACM Workshop on Interactive 3D Grahpics, Association for Computing Machinery, 1986, pp. 77-87.

5. Jean-Marie Hullot. *Interface Builder.* Santa Barbara, CA, 1987.

6. Myron W. Krueger. *Artificial Reality.* Addison-Wesley, Reading, MA, 1983.

7. Paul McAvinney. U.S. Patent No. 4,746,770; Method and Apparatus for Isolating and Manipulating Graphic Objects On Computer Video Monitor. May 24, 1988.

8. Brad A. Myers. Encapsulating Interactive Behaviors. Human Factors in Computing Systems, SIGCHI'89, Austin, TX, April, 1989, pp. (to appear).

9. NeXT, Inc. *Interface Builder.* Palo Alto, CA, 1988. (online, preliminary documentation).

10. Dean Rubine. The Automatic Recognition of Gestures. (thesis proposal, Carnegie Mellon University School of Computer Science).

11. Kurt J. Schmucker. *Object-oriented programming for the Macintosh.* Hayden Book Co., Hasbrouck Heights, N.J., 1986.

12. David Weimer and S. K. Ganapathy. A Synthetic Visual Environment With Hand Gesturing and Voice Input. CHI'89 Conference Proceedings, Association for Computing Machinery's Special Interest Group on Computer Human Interaction, 1989, pp. 235-240.

I. Minimum Distance Between a Point and Segment

```
                  X
                  I
                  I
A-----------P-------------------B
```

Parameterize equation of **AB**:

$$\mathbf{V}(k) = (1-k)\mathbf{A} + k\mathbf{B}$$

for $0 \le k \le 1$, and $\mathbf{A} \le \mathbf{V} \le \mathbf{B}$ so that **V** is any point on the segment between **A** and **B**.

Release the constraint on k for the time being, and let *P* be the point nearest **X** on **AB**: $\mathbf{P} = \mathbf{V}(k_p)$.

This gives us Equation 1:

$$\textit{Eqn 1} \quad \mathbf{P} = (1-k_p)\mathbf{A} + k_p\mathbf{B}$$

or, in expanded form:

$$\mathbf{P} = \mathbf{A} - k_p\mathbf{A} + k_p\mathbf{B}$$

We want a line normal to **AB** that passes through **X**. By definition the dot product is zero if $\angle\mathbf{APX} = 90^o$, so for $\mathbf{XP} \perp \mathbf{AP}$ we have:

$$(\mathbf{P}-\mathbf{X})\cdot(\mathbf{P}-\mathbf{A}) = 0$$

Now substitute for **P**:

$$(\mathbf{A} - k_p\mathbf{A} + k_p\mathbf{B} - \mathbf{X})\cdot(-k_p\mathbf{A} + k_p\mathbf{B}) = 0$$

expand terms:

$$(-k_p+k_p^2)(\mathbf{A}\cdot\mathbf{A}) + (-k_p^2+k_p-k_p^2)(\mathbf{A}\cdot\mathbf{B}) + (k_p^2)(\mathbf{B}\cdot\mathbf{B}) + k_p(\mathbf{A}\cdot\mathbf{X}) - k_p(\mathbf{B}\cdot\mathbf{X}) = 0$$

divide through by k_p and simplify:

$$(-1+k_p)(\mathbf{A}\cdot\mathbf{A}) + (-2k_p+1)(\mathbf{A}\cdot\mathbf{B}) + k_p(\mathbf{B}\cdot\mathbf{B}) + (\mathbf{A}\cdot\mathbf{X}) - (\mathbf{B}\cdot\mathbf{X}) = 0$$

arrange terms for easier reduction:

$$-(1-k_p)(\mathbf{A}\cdot\mathbf{A}) + [(-k_p+1)(\mathbf{A}\cdot\mathbf{B}) - k_p(\mathbf{A}\cdot\mathbf{B})] + k_p(\mathbf{B}\cdot\mathbf{B}) + (\mathbf{A}\cdot\mathbf{X}) - (\mathbf{B}\cdot\mathbf{X}) = 0$$

apply distributive property of dot product:

$$(1-k_p)[\mathbf{A}\cdot(\mathbf{B}-\mathbf{A})] + k_p[(\mathbf{B}-\mathbf{A})\cdot\mathbf{B}] = [\mathbf{X}\cdot(\mathbf{B}-\mathbf{A})]$$

collect terms:

$$k_p[[-\mathbf{A}\cdot(\mathbf{B}-\mathbf{A})] + [(\mathbf{B}-\mathbf{A})\cdot\mathbf{B}]] + [\mathbf{A}\cdot(\mathbf{B}-\mathbf{A})] = [\mathbf{X}\cdot(\mathbf{B}-\mathbf{A})]$$

apply distributive property of dot product again:

$$k_p[(\mathbf{B}-\mathbf{A})\cdot(\mathbf{B}-\mathbf{A})] = [(\mathbf{X}-\mathbf{A})\cdot(\mathbf{B}-\mathbf{A})]$$

solve for k_p:

$$\textit{Eqn 2} \quad k_p = \frac{(\mathbf{B}-\mathbf{A})\cdot(\mathbf{X}-\mathbf{A})}{(\mathbf{B}-\mathbf{A})\cdot(\mathbf{B}-\mathbf{A})}$$

Note that if $k_p < 0$, the nearest point to **X** is **A**. If $k_p > 1$, it is **B**. Otherwise solve Eqn 1 with value of k_p from Eqn 2 to get the nearest point.

Appendix B: Sensor Frame UNIX Device Driver Library Functions

Following is a list of the Sensor Frame UNIX device driver C-callable functions:

sf_open(connection)
sf_close(sf_fd)

sf_perror(string)

sf_scale(sf_fd, xmin, xmax, ymin, ymax, zmin, zmax)
sf_query_scale(sf_fd, xmin, xmax, ymin, ymax, zmin, zmax)

sf_enable(sf_fd, types, boolean)
sf_q_enable(sf_fd, types)

sf_queue(sf_fd, types, boolean)
sf_q_queue(sf_fd, types)

sf_poll_once(event_structure)
sf_poll_all(boolean, sf_fd, event_structure)

sf_qtest( )
sf_qread(event_structure)
sf_qflush( )
sf_qadd(event_structure)
sf_qpush(event_structure)

sf_user_input_handler(sf_fd, user_function_address)

sf_filter(sf_fd, filter_structure)
sf_q_filter(sf_fd, filter_id, filter_structure)

sf_toss( )

Appendix C: Contents of Sensor Frame Videotape (VHS Format)

Appendix C consists of a VHS Videotape showing the Following:

- Appendix C-1: The Sensor Frame
- Appendix C-2: The First and Second Prototype Sensor Cubes
- Appendix C-3: The Gesture Based User Interface Prototyping System (GID)
- Appendix C-4: The VideoHarp

Copies of the Videotape are available upon request from Sensor Frame Corporation.